7 DisCoPy

El objetivo de este capítulo es transformar las definiciones matemáticas definidas en los capítulos previos en información que puede ser interpretada mediante código en `Python`.
Para este fin, exploraremos la librería `DisCoPy` (Distributional Compositional Python), introducida en [1] por Bob Coecke, Giovanni di Felice y Alexis Toumi, una herramienta de código abierto para el cómputo de categorías y diagramas de cables.
La instalación de la paquetería `DisCoPy` se realiza con la siguiente línea de código

```
    pip install discopy
```

El código presentado en este capítulo puede consultarse en el siguiente repositorio: `https://github.com/FerFerreyra/TesisFernando.git`

7.1. Categorías

Como mencionamos anteriormente, los dos ingredientes básicos para definir una categoría $\mathcal{C}$ son una colección de objetos $\mathcal{C}_0$ y un conjunto de morfismos $\mathcal{C}_1$ que satisfacen las propiedades mencionadas con anterioridad.
En el contexto de `Python`, consideramos dos clases: la clase `Ob`, que contiene los objetos de la categoría, y la clase `Flecha`, que contiene los morfismos de la categoría, con los atributos `dom` y `cod` junto a dos métodos: `then` que efectúa la composición de dos flechas e `id` que genera la flecha identidad para cada objeto.
Antes de continuar, para señalar la naturalidad de introducir el concepto de categorías en el lenguaje de programación `Python`, definimos la categoría **Pyth** cuyos objetos son la clase de todos los tipos de `Python` y cuyas flechas son las clase de todas las funciones de `Python`.[1]
El esqueleto de la implementación de una categoría en `Python` está dado por

```
class Ob: ...

class Flecha:
    dom: Ob
```

[1] Notemos que determinar qué es que dos funciones de `Python` sean iguales no es un concepto claro, incluso asumiendo que dos funciones son iguales si y sólo si producen los mismos resultados dado cualquier `input`, entonces verificar la asociatividad y propiedad de la identidad podrían fallar en programas no terminales, o ante la presencia del tipo `Undefined` o `None`. Sin embargo, no todo está perdido y podemos considerar los morfismos de **Pyth** exluyendo casos problemáticos.

```
    cod: Ob

    @staticmethod
    def id(x:Ob) -> Flecha : ...
    def then(self, other: Flecha) -> Flecha : ...
```

A continuación, proveemos de la implementación de una categoría libre en `DisCoPy`, recordemos que tiene como ingredientes básicos los objetos y generadores, o cajas, que forman los morfismos en la categoría. Comencemos con la clase `Ob` los cuales basta inicializar nombrándolos.

```
class Ob:
    def __init__(self, name):
        self.name = name
```

Continuamos con la clase `Arrow` que corresponden a los morfismos de la categoría.

```
class Arrow(Composable[Ob]):
    def __init__(self, inside: tuple[Box, ...], dom: Ob , cod: Ob,
    _scan: bool = True) -> None:

    @classmethod
    def id(cls: Type[Arrow], dom: Optional[Ob] = None) -> Arrow:

    def then(self, *others: Arrow) -> Arrow:
```

Así cada morfismo es representado como una lista de cajas con un dominio y codominio. Notemos que si `_scan = True`, entonces verificamos que la composición de las cajas tenga sentido (es decir, que el codominio de una caja coincida con el dominio de la siguiente).
Y, finalizamos, con la clase `Box` de los generadores

```
class Box(Arrow):
    def __init__(self, name, dom, cod):
        self.name, self.dom, self.cod = name, dom, cod
        Arrow.__init__(self, dom, cod, [self], _scan=False)
```

Podemos definir, por ejemplo, la categoría con los siguiente objetos y morfismos

```
from discopy.cat import Ob, Box
w, x, y, z = map(Ob, "wxyz")
h, f, g = Box('h', w, x), Box('f', x, y), Box('g', y, z)
```

Es importante resaltar que la composición de funciones, `f.then(g)` o `h.then(f).then(g)` se representa como `f>>g` y `h >> f >> g`.
Con esta implementación, podemos representar la gramática del ejemplo 1 de la siguiente forma

```
s0, x0, x1, x2, x3, s1 = map(Ob, "s0 x0 x1 x2 x3 s1".split())
A,B = Box('A', s0, x0), Box('B', s0, x0)
conoce = Box('conoce', x0, x1)
a = Box('a', x1, x2)
C,D = Box('C', x2, x3), Box('D', x2, x3)
quien = Box('quien', x3, x0)
_ = Box('.', x3, s1)
gramatica = [A, B, conoce, a, C, D, quien, _]
```

Como es esperable, queremos determinar cuando una oración es válida en la gramática, así que definimos la siguiente función.

```
def valida_gramatical(oracion, gramatica):
    flecha = Arrow.id(s0)
    bandera = True
    oracion = oracion.split() + ['.']
    for palabra in oracion:
        es_Box = False
        for Box in gramatica:
            if Box.name == palabra:
                try:
                    flecha = flecha.then(Box)
                    es_Box = True
                    break
                except AxiomError:
                    bandera = False
                    break
        if not es_Box:
            bandera = False
            break
    return bandera
```

7.2. Categorías monoidales

A pesar de que en la exposición de este trabajo describimos las multicategorías de manera previa a las categorías monoidales, `DisCoPy` define los conceptos al revés.

Los objetos generadores en la categoría monoidal libre se realizan mediante el constructor `Ty`

```
from discopy import cat
class Ty(cat.Ob):
   def __init__(self, *inside: str | cat.Ob):
   def tensor(self, *others: Ty) -> Ty:
```

Es decir, los objetos serán objetos concatenados por la operación binaria tensor, denotada `@`.

Los morfismos, por otra parte, son representados como los diagramas que generan, así están definidos en virtud de la proposición 8 de la siguiente manera

```
class Diagrama(cat.Arrow):
    def __init__(self,dom,cod,boxes,offsets,layers):

    def tensor(self, other: Diagram = None, *others: Diagram) ->
   Diagram:

    @classmethod
    def id(cls: Type[Diagrama], dom: Optional[Ob] = None) -> Diagram
   :

    def then(self, *others: Diagram) -> Diagram:
```

donde notamos que los métodos `tensor` y `then` corresponden a la composición paralela y secuencial respectivamente.
El último ingrediente para definir una categoría monoidal libre son los generadores, así está la siguiente implementación de la clase `Box`.

```
    class Box(cat.Box, Diagram):
        def __init__(self,name,dom,cod, **params):
```

Implementemos la siguiente categoría monoidal libre, cuyos objetos son

```
from discopy.monoidal import Ty, Box, Diagram

x = Ty("x")
y, z, w = Ty(*"yzw")
```

y con los siguientes morfismos generadores

```
    f, g, h = Box('f', x, y), Box('g', z, w), Box('h', y, z)
```

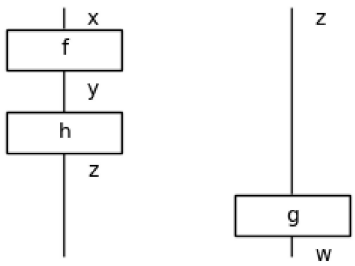
Recordemos que los objetos en la categoría monoidal son "cadenas" de objetos generadores, así si `a,b` son objetos, denotamos como `a @ b` su producto tensorial. De tal manera que su composición secuencial y paralela se representa de la siguiente manera.

```
    f.then(h)
    f @ g
```

La clase de morfismos cuenta con el método `draw` que presenta los diagramas de cables de los morfismos en posición general. Por ejemplo, la siguiente línea de código

```
    (f.then(h) @ g).draw(figsize=(5, 5))
```

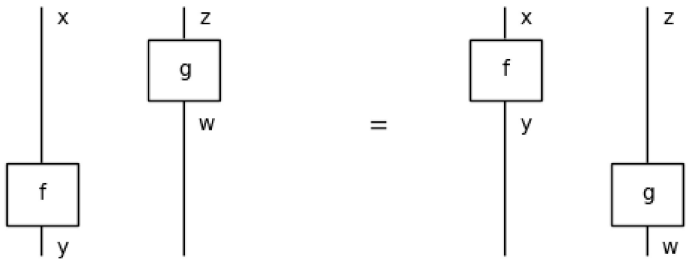
produce el siguiente diagrama



Incluso es posible representar ecuaciones del cáclculo gráfico, por ejemplo, la ley de intercambio se obtiene de la siguiente forma.

```
Equation(
    (f @ g).interchange(0, 1), (f @ g),
    symbol="$=$").draw(figsize=(5, 2))
```

y produce el siguiente diagrama



Como mencionamos en el capítulo 4, las gramáticas monoidales resultan muchas veces poco útiles computacionalmente; sin embargo, esta clase de `DisCoPy` es sumamente eficiente para implementar el resto de gramáticas.

7.3. Multicategorías

Dentro de `DisCoPy`, las multicategorías se encuentran definidas dentro del módulo `grammar.cfg`. Los objetos son implementados con el mismo constructor que las categorías monoidales, pues recordamos que queremos que el dominio de las flechas sea una cadena de objetos.
La diferencia principal son la forma de los morfismos, tal como los definimos en el capítulo dos, son árboles cuya implementación es la siguiente en la clase `Tree`:

```
class Tree:
    def __init__(self, root: Rule, *branches: Tree):

    @staticmethod
    def id(dom):
        return Id(dom)

    def __call__(self, *others) -> Tree:
```

donde el constructor de `Tree` recibe una raíz y sus hojas, junto a árboles tales que las composiciones en las raíces están bien definidas. La composición de árboles se implementa mediante el método `call`.
Por otro lado, los generadores de una multicategoría libre se implementan mediante la clase `Rule`

```
    class Rule(Tree, thue.Rule):
        def __init__(self, dom: monoidal.Ty, cod: monoidal.Ty, name:
    str = None):
```

donde el constructor verifica que la longitud de `cod` es uno, así un elemento de `Rule` es una raíz con sus ramas. Consideremos la implementación de la siguiente multicategoría libre

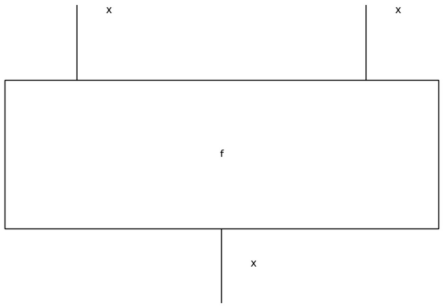
```
from discopy.grammar.cfg import Ty, Tree, Rule

x, y = Ty('x'), Ty('y')
f, g, h = Rule(x @ x, x, name='f'), Rule(x @ y, x, name='g'), Rule(y
    @ x, x, name='h')
```

Tal como la clase `Diagrama`, la clase `Tree` tiene asociado un método para generar los diagramas asociados a los morfismos. La siguiente línea

```
f.to_diagram().foliation().draw()
```

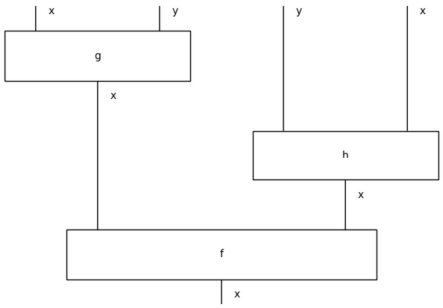
produce el siguiente diagrama



Recordemos que la composición en `Tree` ya no se da por medio del método `then`, sino con `__call__`. Un ejemplo de composición es el siguiente:

```
    f(g, h).to_diagram().foliation().draw()
```

que produce el siguiente diagrama



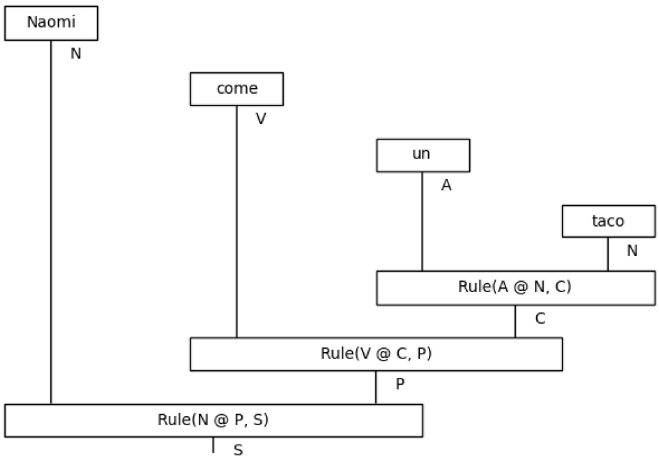
Antes de poder realizar derivaciones en `Tree`, necesitamos definir la clase `Word`

```
class Word(thue.Word, Rule):
     def __init__(self, name: str, cod: monoidal.Ty, dom: monoidal.
   Ty = Ty(),**params):
```

que corresponde a una raíz con una sola hoja, que servirá para poder generar el léxico. Con lo anterior, podemos realizar el código del ejemplo 7 de la siguiente forma:

```
n, v, a = Ty('N'), Ty('V'), Ty('A')
c, p, s = Ty('C'), Ty('P'), Ty('S')
Naomi, come = Word('Naomi', n), Word('come', v)
un, taco = Word('un', a), Word('taco', n)
C, P = Rule(a @ n, c), Rule(v @ c, p)
S = Rule(n @ p, s)
sentence = S(Naomi, P(come, C(un, taco)))
sentence.to_diagram().foliation().draw()
```

y produce el siguiente árbol



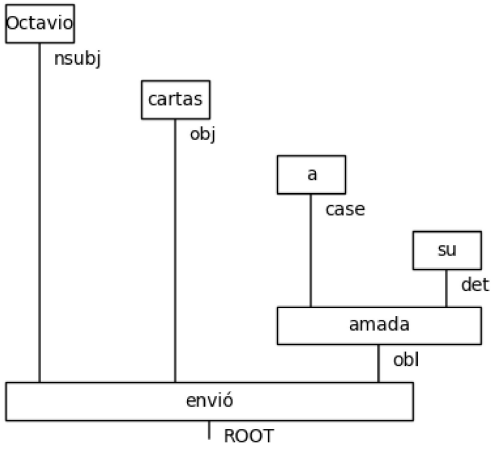
Por último, es importante mencionar que en el módulo `discopy.grammar` de `DisCoPy` se tiene la función `from_spacy` que permite generar árboles de derivación de oraciones con base en el análisis sintáctico realizado por la librería para el procesamiento de lenguaje natural, `spaCy`. Afortunadamente, la librería es capaz de analizar oraciones en español, así que veamos el árbol obtenido al analizar la oración 10.

```
from discopy.grammar.dependency import from_spacy
import spacy

nlp = spacy.load("es_core_news_sm")
doc = nlp("Octavio envio cartas a su amada")

from_spacy(doc).to_diagram().draw(figsize=(4, 4))
```

que produce el siguiente árbol de derivación



7.4. Categorías bicerradas

Las categorías bicerradas se implementa en el módulo `discopy.closed`. Recordemos que los objetos de una categoría bicerrada son de la forma $a, a/b$, $a\backslash b$ y $a \otimes b$. Así, los objetos se implementan de la siguiente forma

```
class Ty(monoidal.Ty): ...

class Over(Ty):
    def __init__(self, left=None, right=None): ...

class Under(Ty):
    def __init__(self, left=None, right=None): ...
```

Donde un objeto `Over(x,y)` representado como `x << y`, es el objeto x/y; mientras que `Under(x,y`, representado como `x >> y` es el objeto $x\backslash y$. Recordemos que la categoría bicerrada se define como la categoría monoidal obtenida por los generadores añadiendo el `curry` y `uncurry` de cada función. Entonces implementamos los morfismos de la siguiente manera:

```
class Diagram(monoidal.Diagram)
    def curry(self, n_wires=1, left=False) -> Diagram : ...

    def uncurry(self) -> Diagram : ...

    class Id(monoidal.Id, Diagram):

    class Box(monoidal.Box, Diagram):
```

donde los argumentos de `curry` son un morfismo, el número de elementos a "currificar" y si es izquierdo o derecho. Tanto `curry` como `uncurry` se definen como funciones sobre los generadores, es decir, sobre `Box`.
Podemos entonces considerar los siguientes objetos y generadores de una categoría bicerrada

```
from discopy.closed import Ty, Box, Curry

x, y, z = map(Ty, "xyz")
f, g, h = Box('f', x, z << y), Box('g', x @ y, z), Box('h', y, x >>
    z)
```

Entonces, podemos aplicar `curry` y `uncurry` de la siguiente manera

```
f.uncurry()
g.curry()
h.uncurry(left=False)
```

Para definir las gramáticas categoriales, *DisCoPy* tiene implementado el módulo `discopy.grammar.categorial` que cuenta con las funciones correspondientes a las reglas de inferencia. A continuación presentamos el código y derivación obtenida de los ejemplos 10 y 12.

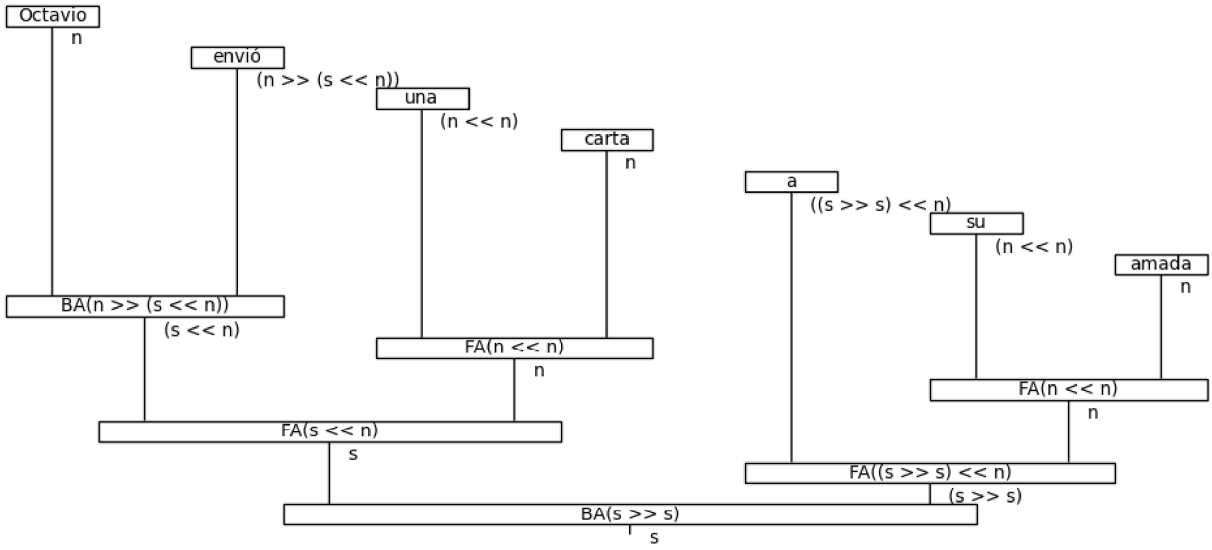
```
from discopy.closed import Ty
from discopy.grammar.categorial import Word, BA, FA
```

```

s, n = map(Ty, 'sn')

Octavio = Word('Octavio', n)
envio = Word('envio', n >> (s << n))
una = Word('una', n << n)
carta = Word('carta', n)
a = Word('a', (s >> s) << n)
su = Word('su', n << n)
amada = Word('amada', n)

sentence = Octavio @ envio @ una @ carta @ a @ su @ amada\
    >> BA(n >> (s << n)) @ FA(n << n) @ ((s >> s) << n) @ FA(n << n)
   \
    >> FA(s << n) @ FA((s >> s) << n)\
    >> BA(s >> s)
sentence.draw()
```

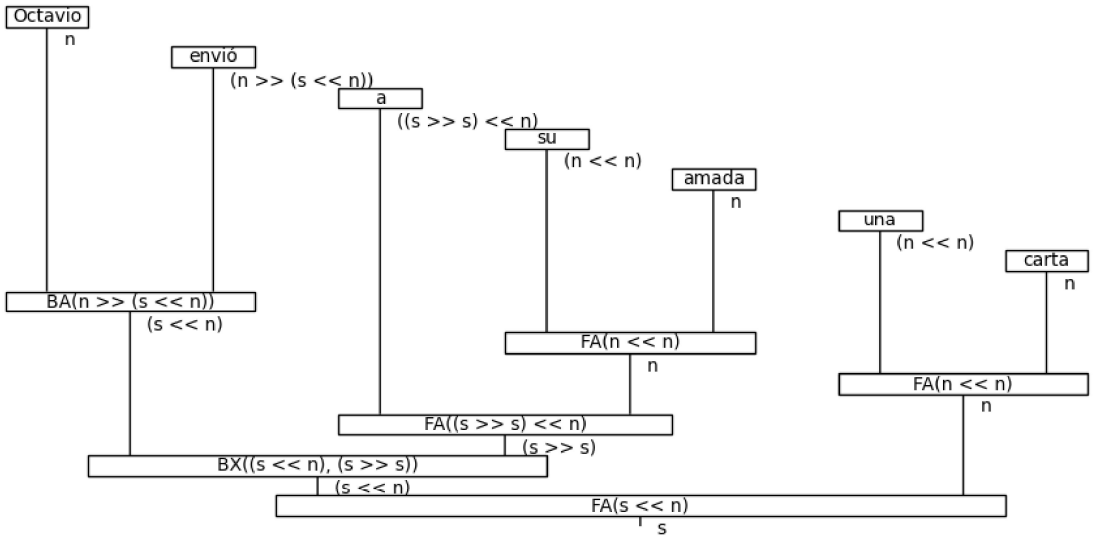



```
from discopy.closed import Ty
from discopy.grammar.categorial import Word, BA, FA, FX, BX

s, n = map(Ty, 'sn')

Octavio = Word('Octavio', n)
envio = Word('envio', n >> (s << n))
una = Word('una', n << n)
carta = Word('carta', n)
a = Word('a', (s >> s) << n)
su = Word('su', n << n)
amada = Word('amada', n)
```

```
13
14 sentence = Octavio @ envio @ a @ su @ amada @ una @ carta\
15              >> BA(n >> (s << n)) @ ((s >> s) << n) @ FA(n << n) @ FA
   (n << n)\
16              >> (s << n) @ FA((s >> s) << n) @ n\
17              >> BX(s << n, s >> s) @ n\
18              >> FA(s << n)
19
20 sentence.draw()
```



7.5. Categorías rígidas

Para la implementación de categorías rígidas, `DisCoPy` utiliza la siguiente convención. Dado un conjuntos de objetos G_0, consideramos los objetos en la categoría rígida libre de la forma $(x, z) \in G_0 \times \mathbb{Z}$ donde definimos que $(x, z)^r = (x, z + 1)$ y $(x, z)^l = (x, z - 1)$.
Notemos que esta definición es completamente compatible con la exhibida en el capítulo 5 pues tenemos la siguiente asignación: dado un elemento $x \in G_0$.

$$x^{\overbrace{r \cdots r}^{n-veces}} \mapsto (x, n) \qquad x \mapsto (x, 0) \qquad x^{\overbrace{l \cdots l}^{n-veces}} \mapsto (x, -n)$$

Así, primero se define la clase de tipos básicos

```
1 class Ob(cat.Ob):
2     def __init__(self, name, z=0): ...
3
4     def l(self) -> Ob: ...
```

```
5
6       def r(self) -> Ob: ...
```

con ello definimos lo elemento dentro de la categoría rígida libre.

```
1 class Ty(monoidal.Ty, Ob):
2     def __init__(self, *t):
3
4     def r(self) Ty(monoidal.Ty, Ob): ...
5
6     def l(self) Ty(monoidal.Ty, Ob): ...
```

Y los morfismos son de la forma de una categoría monoidal libre, diagramas, generados por `cups` y `caps`.

```
1 class Diagram(monoidal.Diagram):
2     def cups(left, right) -> Diagram: ...
3
4     def caps(left, right) -> Diagram: ...
5
6 class Box(monoidal.Box, Diagram):
7
8 class Id(monoidal.Id, Diagram):
```

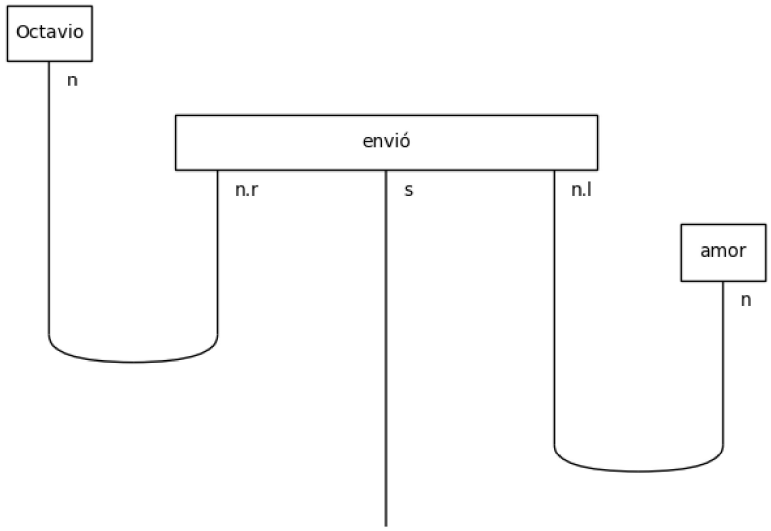
Al igual que con las gramaticas libres de contexto y las gramáticas categoriales, `Discopy` cuenta con la instanciación de la libreria `discopy.rigid` para pregrupos. En ella, podemos definir las evaluaciones y coevaluaciones de la siguiente manera

```
1 from discopy.grammar.pregroup import Ty, Cup, Cap
2
3 n = Ty('n')
4 Cap(n,n.l)
5 Cap(n.r,n)
6 Cup(n.l,n)
7 Cup(n,n.r)
```

Ya que sabemos implementar los pregrupos, hagamos una derivación sencilla.

```
1 from discopy.grammar.pregroup import Ty, Word, Cup
2
3 s, n = Ty('s'), Ty('n')
4 Octavio, amor = Word('Octavio', n), Word('Eliza', n)
5 envio = Word('envio', n.r @ s @ n.l)
6
7 sentence = Octavio @ envio @ amor >> Cup(n, n.r) @ s @ Cup(n.l, n)
8 sentence.draw()
```

que genera el siguiente diagrama



Finalizaremos este capítulo y este texto con el recordatorio que toda categoría rígida es bicerrada y, por lo tanto, existe el `curry`.
A continuación se mostrará la implementación en gramáticas de pregrupos de un analizador basado en `Spacy`. Además, los probaremos con nuestra ya conocida oración "Octavio envió una carta a su amada".

```
from discopy.grammar.pregroup import Ty, Id, Box, Diagram
import spacy

def curry(diagram, n_wires=1, left=False):
  if not n_wires > 0:
    return diagram
  if left: #El curry se realiza por la izquierda
    wires = diagram.dom[:n_wires]
    #Sustituimos los primeros cables por sus caps para realizar el curry
    return Diagram.caps(wires.r, wires) @ Id(diagram.dom[n_wires:]) >> Id(wires.r) @ diagram
  wires = diagram.dom[-n_wires:]
  #Sustituimos los ultimos cables por sus caps para realizar el curry
  return Id(diagram.dom[:-n_wires]) @ Diagram.caps(wires, wires.l) >> diagram @ Id(wires.l)

def find_root(doc):
    for token in doc:
        if token.dep_ == 'ROOT':
            return token

def doc2rigid(word):
  children = word.children
  #Si la raiz no tiene hijos, terminamos
  if not children:
```

```
24      return Box(word.text, Ty(word.dep_), Ty())
25    #Generamos los objetos por los hijos a la izq y a la derecha
26    #y obtenemos la clasificacion sintactica de la palabra en la
       oracion.
27    left = Ty(*[child.dep_ for child in word.lefts])
28    right = Ty(*[child.dep_ for child in word.rights])
29    #Definimos el generador obtenido con la palabra actual
30    box = Box(word.text, left.l @ Ty(word.dep_) @ right.r, Ty(),
31    data=[left, Ty(word.dep_), right])
32    top = curry(curry(box, n_wires=len(left), left=True), n_wires=len(
       right))
33    #Realizamos la llamada recursiva para seguir explorando la oracion
       .
34    bot = Id(Ty()).tensor(*[doc2rigid(child) for child in children])
35    return top >> bot
36
37  def doc2pregroup(doc):
38    root = find_root(doc)
39    return doc2rigid(root)
40
41  nlp = spacy.load("es_core_news_sm")
42  doc = nlp("Octavio envio una carta a su amada")
43
44  doc2pregroup(doc).draw()
```

que produce el siguiente diagrama